Parallel Programming in Data Analysis Software

Alfio Lazzaro
Università degli Studi and INFN, Milano
CERN, Geneva

The case

- In general all methods are based on optimization problems: find a maximum (for example in case of Statistical Significance Maximization or Maximum Likelihood) or a minimum (Expected Prediction Error) of a function
- This is done by numerical algorithms
  - Most commonly used are based on Gradient Descent Methods, which require the calculation of the several derivates of the function
  - Some example of Genetic Algorithm uses
- This procedure can be very slow, depending on the number of free parameters to be determined, the number of input events, and the complexity of the model

Maximum Likelihood Fits

- In Maximum Likelihood fits we have to maximize the likelihood function

$$\mathcal{L} = \frac{e^{-\sum_{j=1}^{s} n_j}}{N!} \prod_{i=1}^{N} \sum_{j=1}^{s} n_j \mathcal{P}_j^i .$$

j species (signals, backgrounds)
n_j number of events for specie *j*
P_j probability
N number total of events to fit

- In general we minimize the Negative Log-Likelihood Function

$$-\ln \mathcal{L} \equiv NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

- The minimization is performed as function of free parameters: n_j number of events, parameters of P_j

Minimization

- The most largely used algorithm for minimization is MINUIT
- MINUIT uses the gradient of the function to find local minimum (MIGRAD), requiring
  - The calculation of the gradient of the function for each free parameter, naively

$$\left.\frac{\partial NLL}{\partial\hat{\theta}}\right|_{\hat{\theta}_0} \approx \frac{\boxed{NLL(\hat{\theta}_0 + \hat{\mathrm{d}})} - \boxed{NLL(\hat{\theta}_0 - \hat{\mathrm{d}})}}{2\hat{\mathrm{d}}}$$

2 per derivate

  - The calculation of the covariance matrix of the free parameters (which means the second order derivates)
- The minimization is done in several steps moving in the direction of the negative gradient value: each step require the calculation of the gradient

Minimization

- In case of NLL function, it requires the calculation of the function for each free parameter in each minimization step

  1. Many free parameters means slow calculation
  2. Remember the definition of *NLL*

  $$NLL = \ln\left(\sum_{j=1}^{s} n_j\right) - \sum_{i=1}^{N}\left(\ln \sum_{j=1}^{s} n_j \mathcal{P}_j^i\right)$$

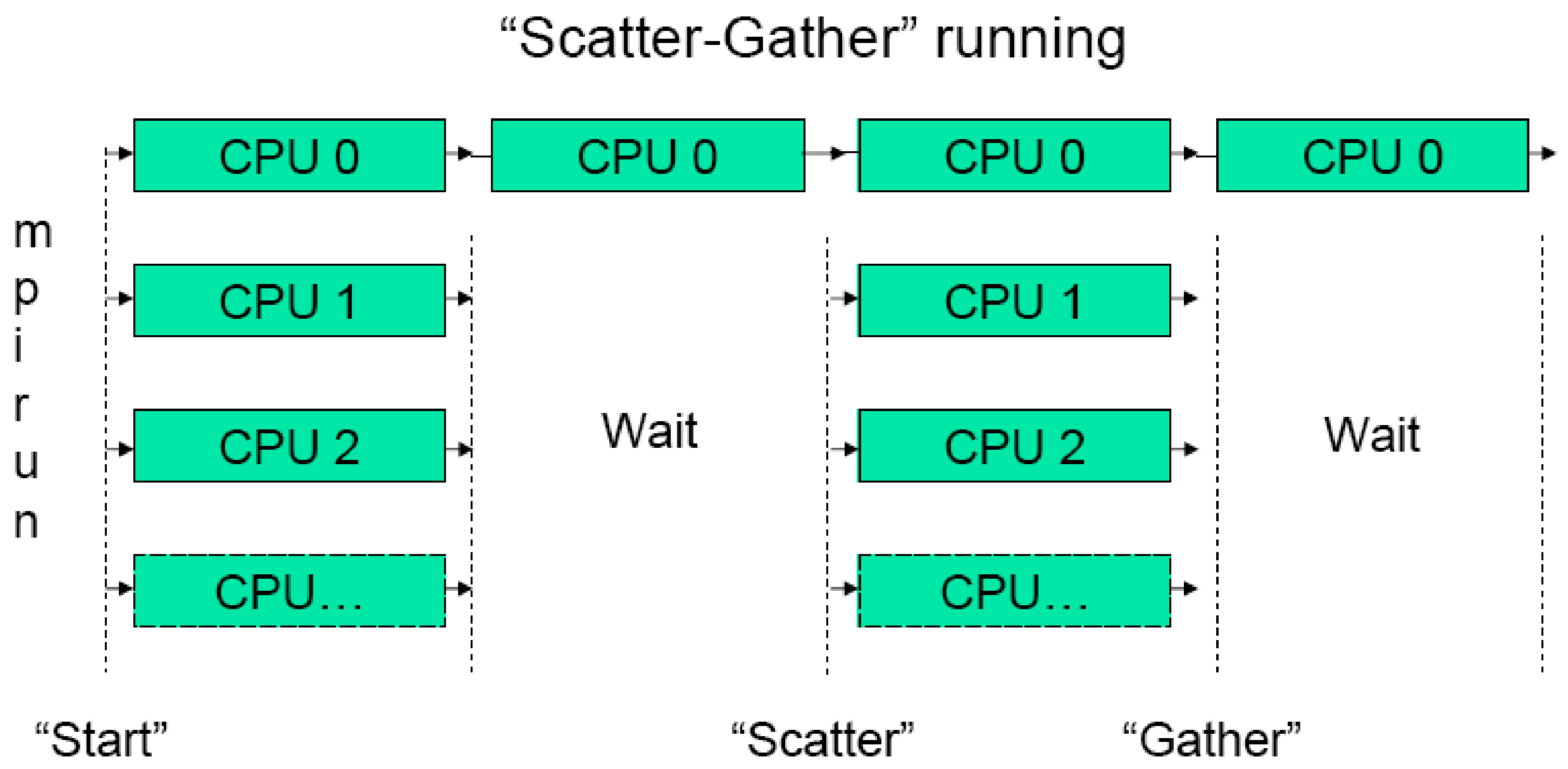
     The computational cost scales with the *N* number of events in the input sample

  3. Note, also, that P_j need to be normalized (calculation of the integral) for each iteration, which can be a very slow procedure if we don't have an analytical function

- Complex fits take several hours (or days)!

Parallelization

- RooFit (Maximum Likelihood fit package) implements the possibility to split the likelihood calculation over different threads (point 2)
  - Likelihood calculation is done on sub-samples
  - Then the results are collected and summed
  - You gain a lot using multi-cores architecture over large data samples, scaling almost with a factor proportional to the number of threads
- However, if you have a lot of free parameters, the bottleneck become the minimization procedure (point 1)
  - Split the derivate calculation over several MPI processes
  - There is not an official implementation of such a algorithm, but some tests done by people in BaBar
  - You can gain almost a factor proportional to the number of processes (almost...)



"Trivial" splitting of the NLL function calculation for NPAR parameters over NCPU CPUs: NPAR/NCPU parameters for each CPU

Example:

NPAR = 10, NCPU = 3

$$\left\{\begin{array}{l}\text{CPU 1 = 4 pars}\\ \text{CPU 2 = 3 pars}\\ \text{CPU 3 = 3 pars}\end{array}\right.$$

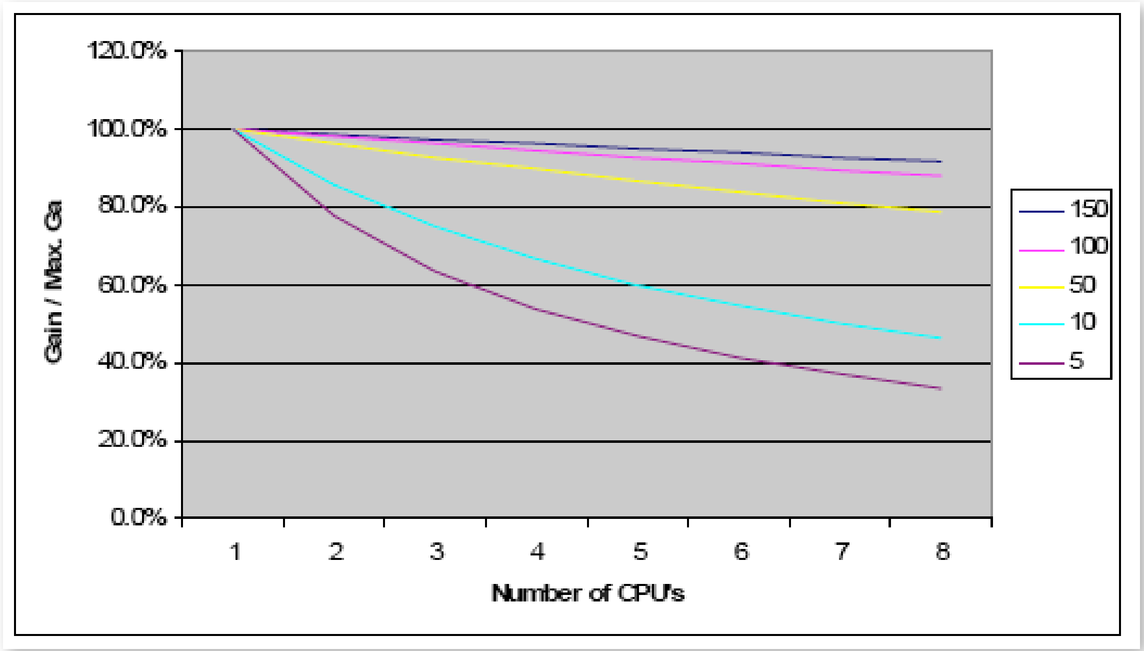
Max threads = NPAR

Parallelization

It works well in case of large number of parameters

Gain ~ NCPU*(NPAR + 2) / (NPAR + 2*NCPU)

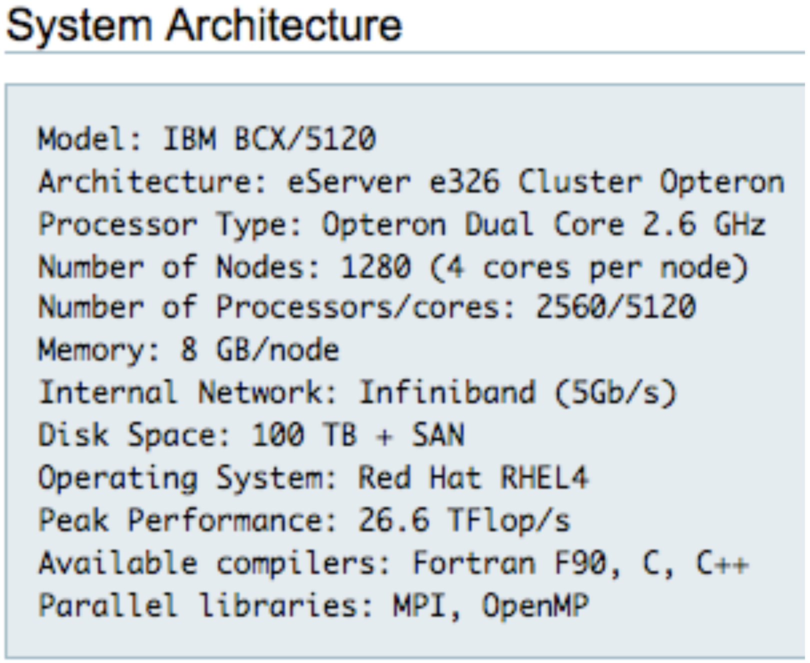
Max. Gain = NCPU



From Brian Meadows talk at RooFit Mini Workshop @ SLAC (December 2007):
http://www.slac.stanford.edu/BFROOT/www/doc/Workshops/2007/BaBar_RooFit/Agenda.html

Parallelization - Work

- Based on ROOT 5.20 and MINUIT2.
    - Lorenzo already implemented a OpenMP version of MINUIT2
- Hybrid of the likelihood calculation using multi-threads minimization process using MPI ==> higher gain in case of multi-cores/MPI case
- Work done using resources at CINECA HPC Center in Italy (Bologna): https://hpc.cineca.it/

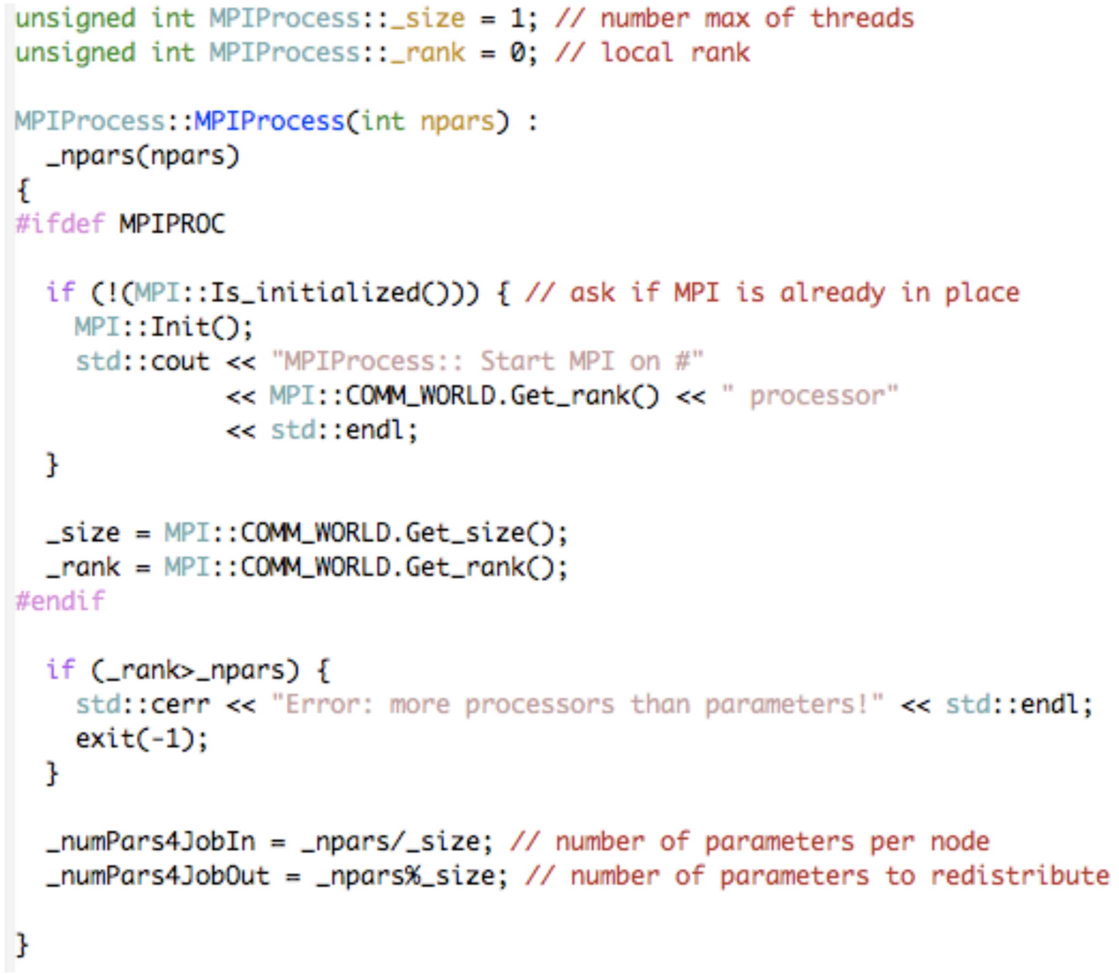


System Architecture

```
Model: IBM BCX/5120
Architecture: eServer e326 Cluster Opteron
Processor Type: Opteron Dual Core 2.6 GHz
Number of Nodes: 1280 (4 cores per node)
Number of Processors/cores: 2560/5120
Memory: 8 GB/node
Internal Network: Infiniband (5Gb/s)
Disk Space: 100 TB + SAN
Operating System: Red Hat RHEL4
Peak Performance: 26.6 TFlop/s
Available compilers: Fortran F90, C, C++
Parallel libraries: MPI, OpenMP
```

Details -- Ctor

- Added a new class in the MINUIT2 package which deals with MPI: MPIProcess
- Inserted define directives to switch off MPI at level of compilation

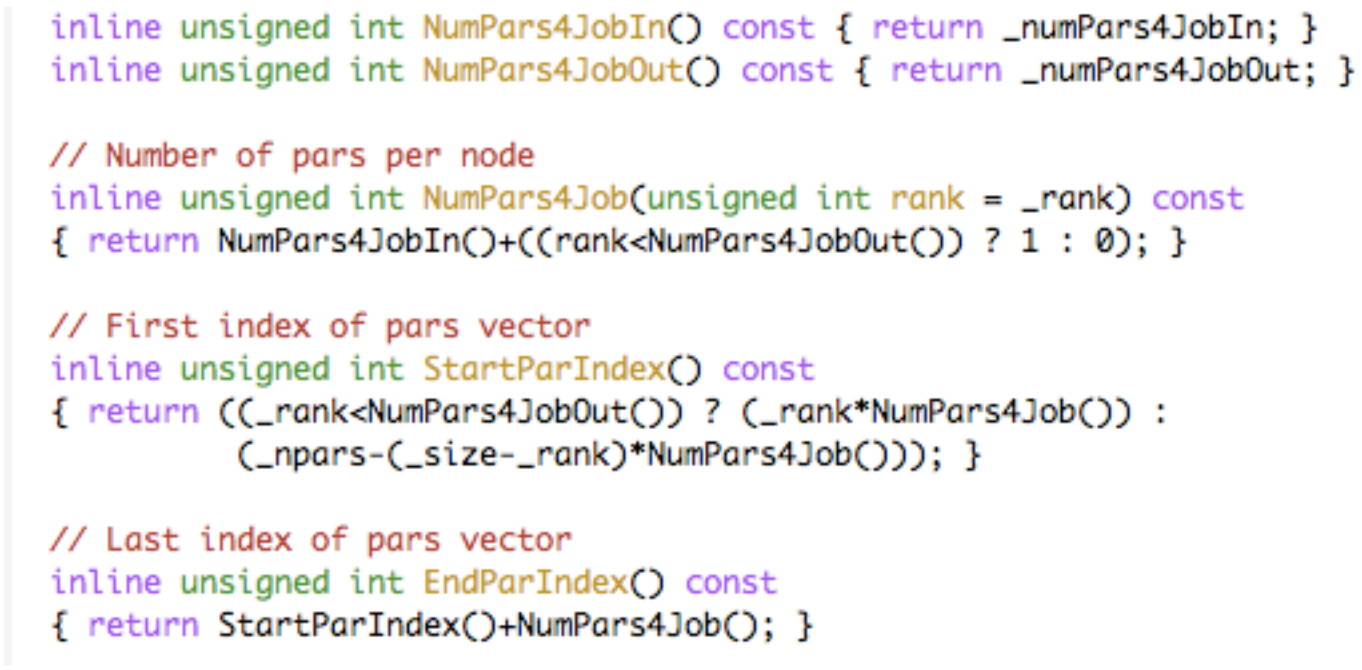


MPI

```
unsigned int MPIProcess::_size = 1; // number max of threads
unsigned int MPIProcess::_rank = 0; // local rank

MPIProcess::MPIProcess(int npars) :
  _npars(npars)
{
#ifdef MPIPROC

  if (!(MPI::Is_initialized())) { // ask if MPI is already in place
    MPI::Init();
    std::cout << "MPIProcess:: Start MPI on #"
              << MPI::COMM_WORLD.Get_rank() << " processor"
              << std::endl;
  }

  _size = MPI::COMM_WORLD.Get_size();
  _rank = MPI::COMM_WORLD.Get_rank();
#endif

  if (_rank>_npars) {
    std::cerr << "Error: more processors than parameters!" << std::endl;
    exit(-1);
  }

  _numPars4JobIn = _npars/_size; // number of parameters per node
  _numPars4JobOut = _npars%_size; // number of parameters to redistribute

}
```

Pars Distribution per Node

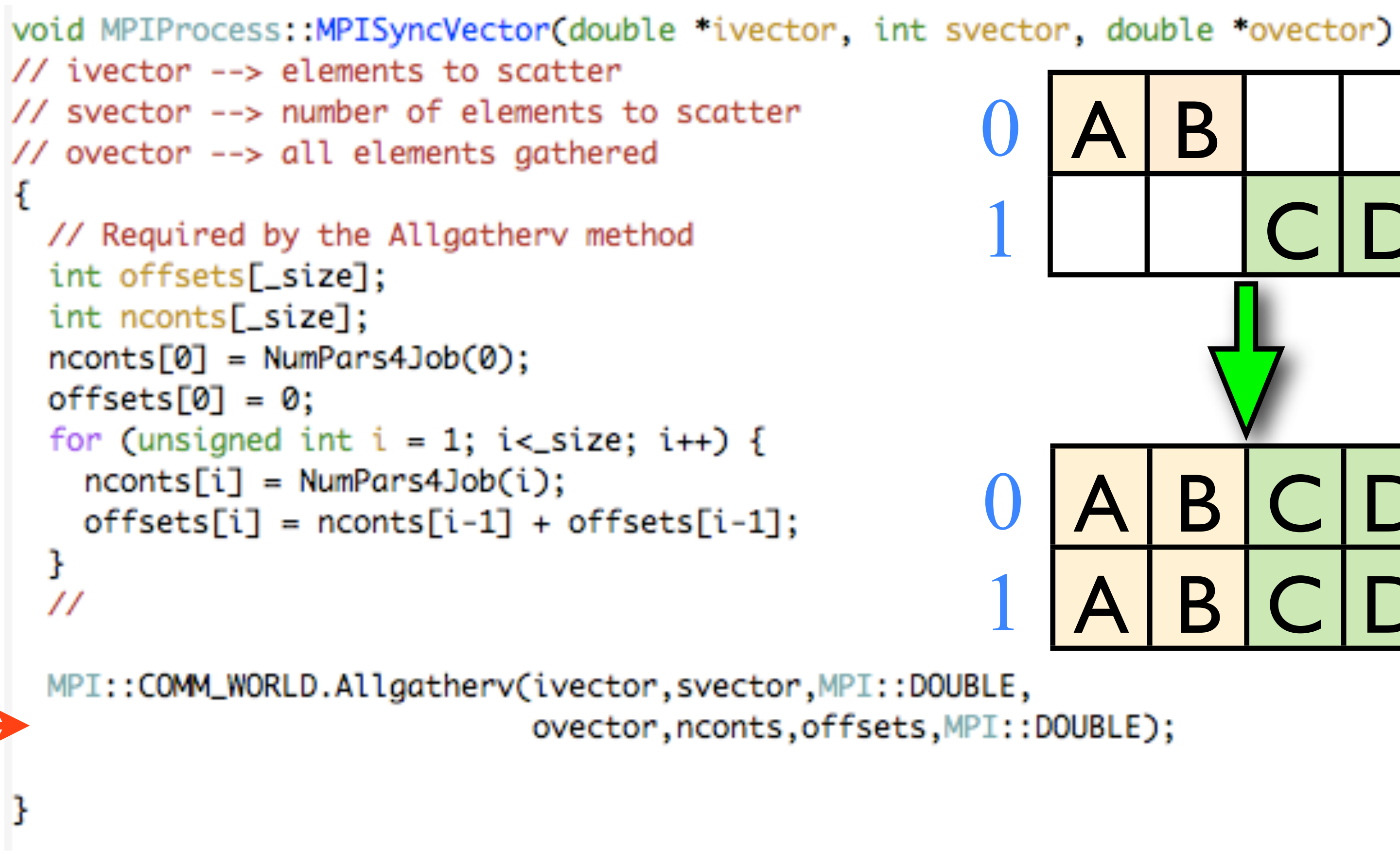
- Results for derivates organized in a vector of doubles
- Index of the vector per node:



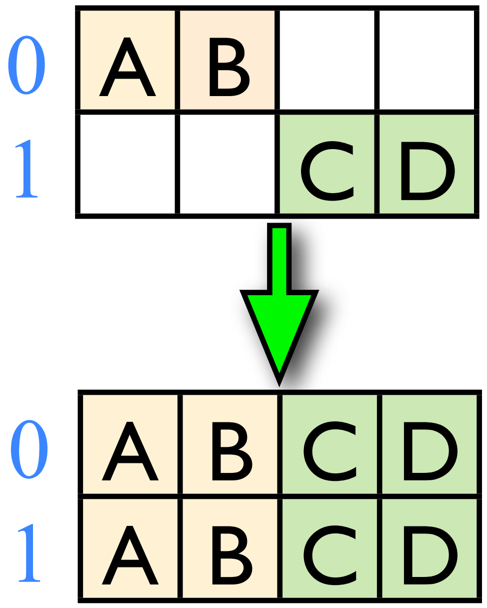
```
inline unsigned int NumPars4JobIn() const { return _numPars4JobIn; }
inline unsigned int NumPars4JobOut() const { return _numPars4JobOut; }

// Number of pars per node
inline unsigned int NumPars4Job(unsigned int rank = _rank) const
{ return NumPars4JobIn()+((rank<NumPars4JobOut()) ? 1 : 0); }

// First index of pars vector
inline unsigned int StartParIndex() const
{ return ((_rank<NumPars4JobOut()) ? (_rank*NumPars4Job()) :
          (_npars-(_size-_rank)*NumPars4Job())); }

// Last index of pars vector
inline unsigned int EndParIndex() const
{ return StartParIndex()+NumPars4Job(); }
```

Synchronization

- Each node calculates his group of derivates
- Each node scatters the results to all other nodes
  - At the end all nodes have the results for all derivates

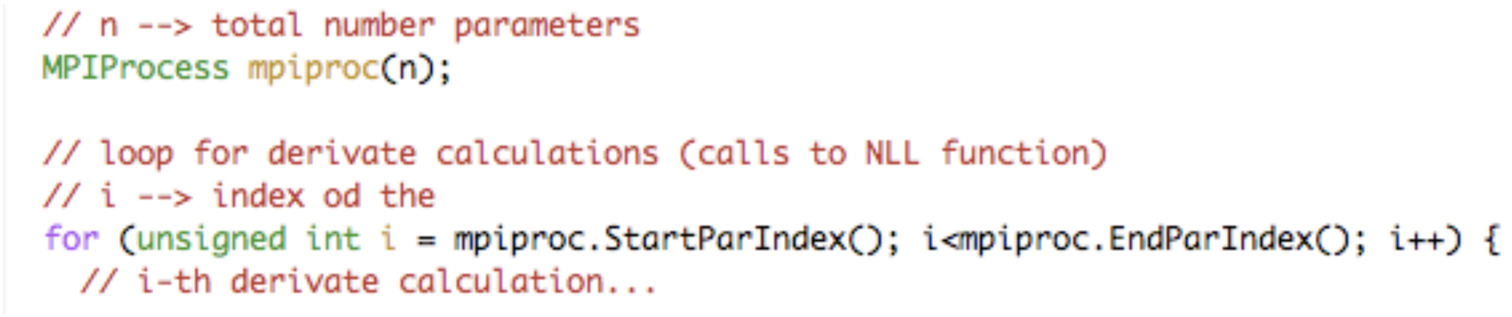


```
void MPIProcess::MPISyncVector(double *ivector, int svector, double *ovector)
// ivector --> elements to scatter
// svector --> number of elements to scatter
// ovector --> all elements gathered
{
  // Required by the Allgatherv method
  int offsets[_size];
  int nconts[_size];
  nconts[0] = NumPars4Job(0);
  offsets[0] = 0;
  for (unsigned int i = 1; i<_size; i++) {
    nconts[i] = NumPars4Job(i);
    offsets[i] = nconts[i-1] + offsets[i-1];
  }
  //

  MPI::COMM_WORLD.Allgatherv(ivector,svector,MPI::DOUBLE,
                             ovector,nconts,offsets,MPI::DOUBLE);

}
```



Call in MINUIT2

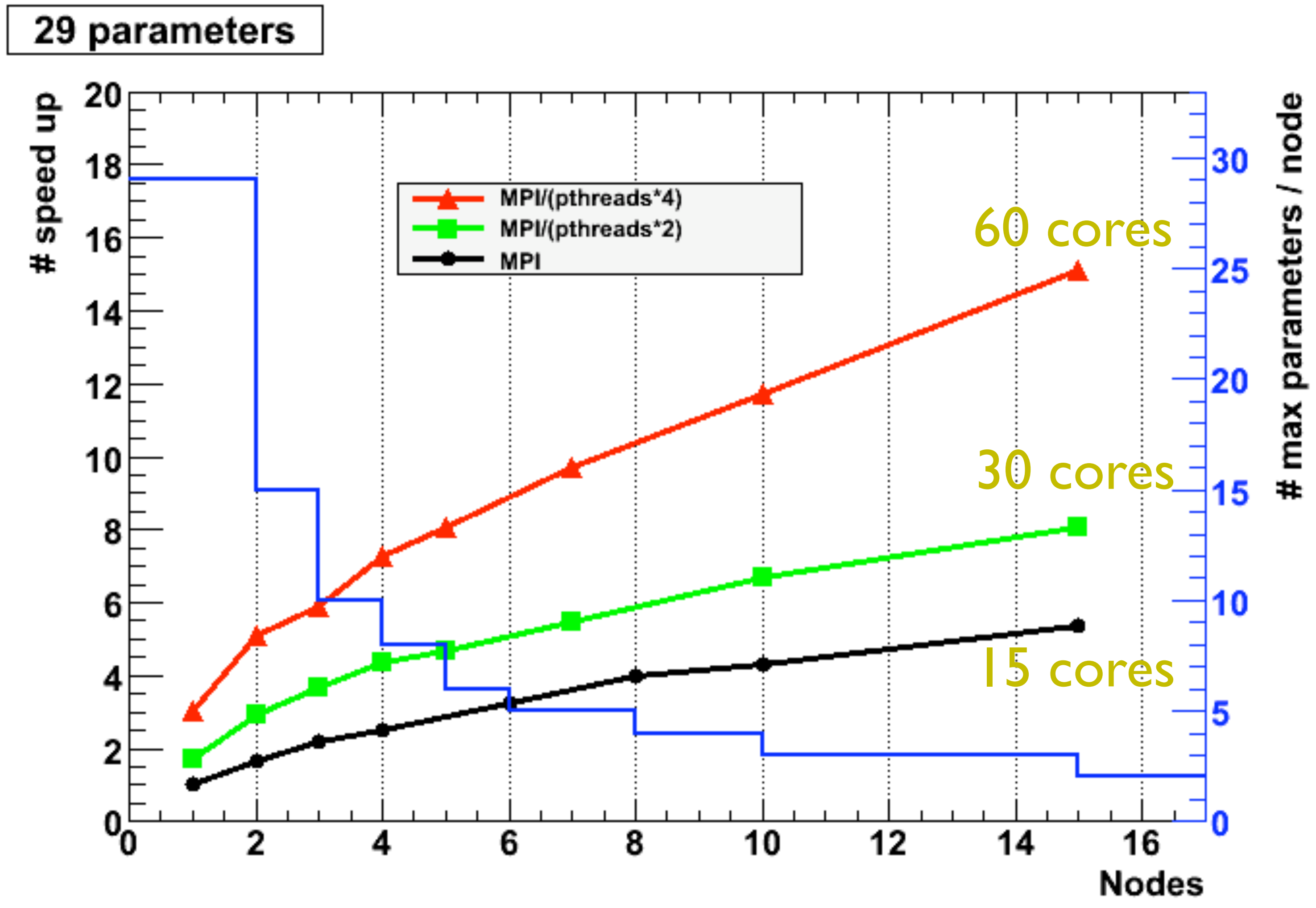
- MPIProcess declared in the class for Gradient calculation (Numerical2PGradientCalculator)
- Each node calculates his group of derivates



```
// n --> total number parameters
MPIProcess mpiproc(n);

// loop for derivate calculations (calls to NLL function)
// i --> index od the
for (unsigned int i = mpiproc.StartParIndex(); i<mpiproc.EndParIndex(); i++) {
  // i-th derivate calculation...
```

- Synchronization: At end each node has all derivate results and can do the remaining part of the (serial) code
  - all nodes do the same serial part
- The full procedure is repeated for each step in the minimization processes

Parallelization - Example



Parallelization - To do

- In some case the bottleneck is the integral calculation (in Dalitz plot analysis we have integral in several variables, which is very slow to compute)
  - There is not parallel implementation of the normalization integral calculation
  - Needs a general infrastructure in RooFit
- Better balance of the parameters over the different nodes:
  - some parameters are slower than other (they involve long integral calculations)
- Possibility to split also the likelihood calculation using MPI (partial implementation using a cartesian topology)
- Test on "conventional" HEP cluster (high latency)

Other example of HPC

- Selection of events applying different cuts: PROOF project, implemented in ROOT:
  - allowing transparent analysis of large sets of ROOT files in parallel on compute clusters or multi-core computers, splitting the data sample
- Bagging and Boosting Technique (Boost Decision Tree) can be very CPU-time consuming
  - Several variables on several events
  - Trees are almost independent, they can split in a parallel architecture
- Neural Networks can implemented on parallel architectures
- In HEP community there is not mention of Trees and Neural Networks (as far as I know) using HPC

Conclusions

- Work is ongoing for NLL parallelization:
    - require changes in RooFit and MINUIT2
- HEP is entering in the precise measurements era:
    - Huge quantity of data available
    - More data will be available soon from LHC experiments
- Most of the data analysis techniques are very CPU-time consuming
    - We can benefit using parallel version of the code
    - In most of the case easy to implement
- Few implementations already available in HEP, but a lot a work still to do for a full parallelization
- Hardware and software (compilers, MPI, ...) technologies ready to go!
    - Possibility to use massive parallel solution: GPUs, FPGA, ...